Equivalence between PDAs and CFGs

In this section we describe how the notions of PDAs and CFGs are equivalent.

CFG to PDA

> Given a context free language, we can construct a PDA that recognizes the same language.

Construction

The idea is somewhat straightforward.

- There is a main chain of 4 states, with auxiliary states that form "loops".
- We start by placing the "end of stack" symbol and the start variable on the stack.
- At any given time we can either:
  - match the top of the stack with the next input, and hence advance the input, or
  - replace the nonterminal at the top of the stack with a derivation of it. This is done in a sequence of states that ends up forming a loop.
- The only accept state can be reached only by popping the "end of stack" symbol. This can only be seen if a derivation of the start variable was successfully matched by the input.

Here is the basic picture:

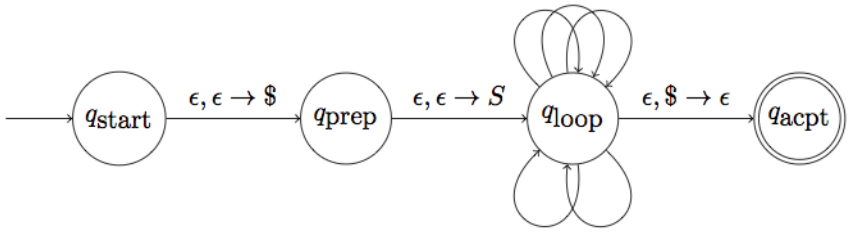

Figure 1: Pushdown Automaton for a CFG

The "loops" around q_{loop} correspond to the production rules in the grammar. Each loop is a shorthand for a sequence of states. For instance if we have a production rule like $S \to aTb$, this would produce the following "loop":

We usually skip the intermediate states in the process and just write this path as a self-loop with transition $\epsilon, S \to aTb$.

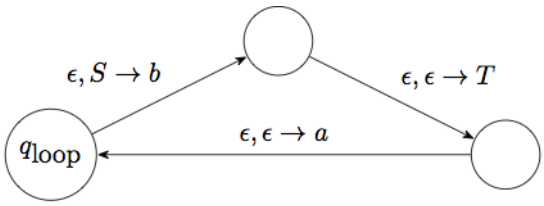

Figure 2: Loop for rule $S \to aTb$

The graph also contains self-loops on q_{loop} with transitions $t, t \to \epsilon$, for each terminal symbol t. These self-loops allow us to advance the input every time we get a terminal at the top of the stack.

This completes the construction of the PDA.

Proof of Construction

We now want to show that this PDA recognizes the same language as the CFG. Effectively the idea is that moving through the PDA is exactly tracing a left-most derivation in the grammar.

- We essentially start at the q_{loop} state, with S at the top of the stack.
- At any given time, the element at the top of the stack would be either
  - a terminal symbol, meaning our left-most derivation has produced a terminal on its left-most end, which we match by the input and advance, or
  - a non-terminal symbol, representing the currently left-most non-terminal as we progress through a left-most derivation in the language. The terminals that came before it have already been matched to input, and this non-terminal at the stack is the one we would need to work on next in a left-most derivation. This is exactly what the PDA's transitions described above allow us to do.
- The end of the process is when we see the "end of input" symbol, meaning that we have managed to replace all non-terminals with sequences of terminals, which then further got popped off the stack by matching the input. This corresponds exactly to the end of a left-most derivation in CFG producing the string.

An example will make that more clear. Consider the language:

```
S -> aSa | bSb | epsilon
```

In addition to the standard 4 states, it contains:

```
- 2 states that establish the loop $\epsilon,S\to aSa$,
- 2 states that establish the loop $\epsilon,S\to bSb$,
- a self-loop for $\epsilon,S\to \epsilon$,
- the two self-loops $a,a\to\epsilon$ and $b,b\to\epsilon$.
```

Now consider the string abbaabba. Here is how it would be computed by this automaton:

Input	Stack	Step from previous
abbaabba	S$	(moved to loop state)
abbaabba	aSa$	eps,S –> aSa
bbaabba	Sa$	a,a –> eps
bbaabba	bSba$	eps,S –> bSb
baabba	Sba$	b,b –> eps
baabba	bSbba$	eps,S –> bSb
aabba	Sbba$	b,b –> eps
aabba	aSabba$	eps,S –> aSa
abba	Sabba$	a,a –> eps
abba	abba$	eps,S –> eps
bba	bba$	a,a –> eps
ba	ba$	b,b –> eps
a	a$	b,b –> eps
eps	$	a,a –> eps
eps	eps	eps,$ –> eps (moving to accept state)

PDA to CFG

For the converse direction, we want to show that given a PDA we can produce a CFG that recognizes the same language. In order to achieve this we first make some simplifying assumptions about the PDA:

- The PDA empties its stack before accepting a string. We already saw how to do that.
- Every transition in the PDA must push something on the stack, or pop something, but it may not do both. So for a transition $t, \alpha \to \beta$ then exactly one of α, β would have to be ϵ. This is easy to achieve by doing the transition in two steps, first doing the pop and then the push, if they were both occuring. Or if neither is occuring, we break it into two steps where we first do a push of an arbitrary symbol and then promptly pop it back out.
- There is only one accept state.

Now we build a CFG for such an automaton.

Construction

- We have one non-terminal symbol $A_{p,q}$ for each pair of states (p, q). It represents a string that when computed in the PDA starting at state p with empty stack will end (perhaps as one of a set of possible computations) at state q with empty stack.
- The start non-terminal is the one corresponding to the pair of the start state and the accept state. It indicates that we want to start at the start state with empty stack, and end at the accept state, with empty stack. If we can produce a valid derivation of our string from this nonterminal, it would correspond to a computation in the PDA that takes us from the start state to the accept state.

- We add a number of production rules:
  - For each three states p, q, r, there is a rule:

    $$A_{p,q} \to A_{p,r}A_{r,q}$$

    This corresponds to the fact that if the string is split in two halves where the first takes us from p to r and the second takes us from r to q, then the whole string takes us from p to q.
  - For each state p, there is a rule

    $$A_{p,p} \to \epsilon$$

    recognizing the fact that you can go from p to p on no input.
  - For any states $p, q, r, s \in Q$ and stack symbol $t \in \Gamma$ and elements $a, b \in \Sigma_\epsilon$, if $(r, t) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$, then we add a rule:

    $$A_{p,q} \to aA_{r,s}b$$

    This rule indicates that if there is a transition to state r from state p on input t and by pushing t to the stack, and a way to go from state r to state s without an overall change in the stack, expressed via the non-terminal $A_{r,s}$, and finally a transition from s to q with input b and that pops the symbol t from the stack, then the combination of these 3 steps is a way to go from p to q without changing the stack.

Before we proceed to prove the claim, we will look at an example, namely the automaton that recognized the language consisting of a number of x's followed by no more than an equal number of y's. In order to accomodate our conditions above, some states need to be added:

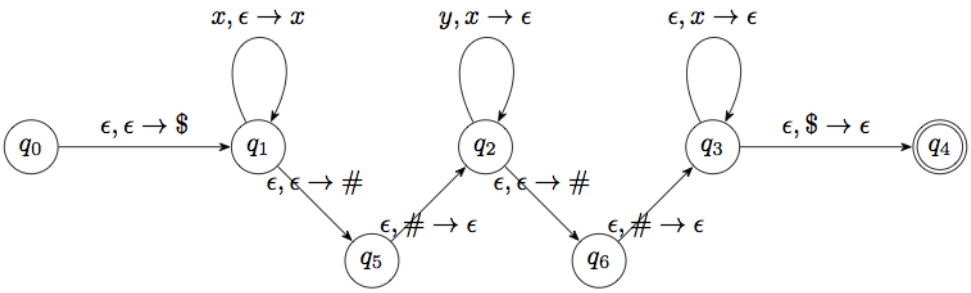

Figure 3: A simple pushdown automaton

We added a "dummy stack symbol" $\#$ to accomodate the cases where we had to represent a transition that did not change the stack.

The corresponding CFG will in theory have up to 30 non-terminals, but many of them are irrelevant. We will present the relevant non-terminals as we build them:

- We start with the start non-terminal $S' = A_{0,4}$.

- Since $(q_1, \$) \in \delta(q_0, \epsilon, \epsilon)$, and $(q_4, \epsilon) \in \delta(q_3, \epsilon, \$)$, we have a rule $A_{0,4} \to \epsilon A_{1,3}\$$. We will call $A_{1,3}$ the "real" start non-terminal, S. So we have the rule $S' \to S\$$. It is this S that should generate the various acceptable strings. Note that this is the only real production rule for S', as no other transitions consume $\$$.
- Next we look at the fact that $(q_1, x) \in \delta(q_1, x, \epsilon)$. This would be "paired up" with a transition of the form $(., \epsilon) \in \delta(., ., x)$. There are exactly two such transitions, $(q_2, \epsilon) \in \delta(q_2, y, x)$ and $(q_3, \epsilon) \in \delta(q_3, \epsilon, x)$. These give rise to the production rules $A_{1,2} \to xA_{1,2}y$ and $A_{1,3} \to xA_{1,3}$. The former corresponds to the fact that a y needs to be matched by an x, the latter to the fact that we could have more xs to begin with.
- The presence of the transitions involving the new dummy stack symbol $\#$ give rise to the rules $A_{1,2} \to A_{5,5}$ and $A_{2,3} \to A_{6,6}$. These boil down to the rules $A_{1,2} \to \epsilon$ and $A_{2,3} \to \epsilon$.
- We also have rules $A_{1,3} \to A_{1,2}A_{2,3}$. Combined with the only rule for $A_{2,3}$ that makes sense, namely $A_{2,3} \to \epsilon$, we get that $A_{1,3} \to A_{1,2}$.

So to sum up, by denoting $S = A_{1,3}$, $T = A_{1,2}$, the language becomes (in reality it has many more extraneous elements that don't lead to anything):

```
S' –> S$
S  –> xSy | T
T  –> xT
```

Check that this grammar will produce the language we expect.

Proof

Now we sketch the proof that the CFG thus produced does recognize the same language as the automaton we started with.

The key claim we need to prove is the following:

> $A_{p,q}$ generates x if and only if x can bring the PDA from state p with empty stack to state q with empty stack.

Before we prove this, let us make an important observation:

> If a string x can bring the PDA from state p with empty stack to state q with empty stack, then it can also bring the PDA from state p with stack symbol t at the top to state q with stack symbol t at the top, without ever popping that symbol on the way. It will simply follow the exact same steps.

Now we prove this claim.

This claim has two directions, we start with the "if" part.

- Suppose $A_{p,q}$ generates x. We want to show that x can bring the PDA from state p with empty stack to state q with empty stack.

- We will do induction on the length of the derivation.
- If the length is 1, meaning a single production rule, then the production rule must have no non-terminals on the right-hand side. The only such possibility is the rule $A_{pp} \to \epsilon$, meaning that $q = p$ and $x = \epsilon$, and it clearly takes p to p. So the base case is proven. Now suppose that we already have established the result for all derivations of at most k steps, and we consider a derivation of $k + 1$ steps. Let us consider what the first step may be. It has two options really.
  - It may be a step $A_{p,q} \to aA_{r,s}b$, where for some stack symbol t we have $(q, \epsilon) \in \delta(s, b, t)$ and $(r, t) \in \delta(p, a, \epsilon)$. In this case our string x would have the form $x = ayb$ and the rest of the derivation is a derivation of y from $A_{r,s}$ in k steps. By the inductive hypothesis, this corresponds to a computation in the PDA from the state r and with an empty stack to the state s and with an empty stack, consuming the string y on the way. Or as we observed earlier, it could do so even if the stack is not empty, and in that case it promises not to change the part of the stack that is already there.

    Now if we start at state p and our string is $x = ayb$, then we read input a and move to the state r while also pushing t to the stack (our transition told us we can do it). We would then follow y to arrive at state s and having t at the stack. Finally, our transitions also told us that we can pop t from the stack to go from s to q. Putting these 3 steps together we find that x has taken us from p with empty stack to q with empty stack. This completes the argument for this case.
  - It may be a step $A_{p,q} \to A_{p,r}A_{r,q}$. Then our string can be separated in two parts $x = yz$ where $A_{p,r}$ derives y and $A_{r,q}$ derives z, by using the subsequent steps in the original derivation. Since those derivations are shorter, the inductive hypothesis tells us that y can take the PDA from p on an empty stack to r on an empty stack, and q can take the PDA from r on an empty stack to q on an empty stack. The combination of these two would then have the desired effect.

Intuitively the idea of this proof is this: The PDA is to start and end with an empty stack as it moves from p to q. If it also has an empty stack at some intermediate point, then we use our second case above to break the problem up into the two subparts. Otherwise, the symbol that is pushed onto the stack as we take our first step from p is not going to be popped out until we do our last step to get to q. This is the kind of situation described by our first case.

Now we want to consider the opposite direction:

- Suppose that the string x manages to transition the PDA from the state p with an empty stack to the state q with an empty stack. We want to show that in fact in our CFG the symbol $A_{p,q}$ could derive x.
- We will do this by induction on the number of steps in the computation.
- First let us suppose there were 0 steps. Then we must have that $p = q$, and we must show that $A_{p,p}$ derives x. Since there were only 0 steps, no input can have been read, meaning that $x = \epsilon$. But we already know the production rule $A_{p,p} \to \epsilon$ in the CFG.

- Now for the inductive step, suppose that the claim is true for any computation that takes no more than k steps, and suppose our computation takes $k+1$ steps.
- The stack starts and ends empty, so we have two cases:
  - If the stack also becomes empty somewhere inbetween, when the computation makes it to state say r, then the string can be split in two parts $x = yz$ where y is the part of the input that was consumed to get us to the state r, and z is what remains. The computation that brought us to r consists of at most k steps, and therefore from the inductive hypothesis we would know that $A_{p,r}$ derives y. Similarly, we would know that $A_{r,q}$ derives z. Putting the two together and prepending the rule $A_{p,q} \to A_{p,r}A_{r,q}$ we get a derivation of the string $x = yz$ from $A_{p,q}$.
  - If the stack does not become empty inbetween, denote by t the element that is pushed onto the stack by the very first step (since the stack is empty, the first step can't be a pop). Then the first transition looks something like $(r, t) \in \delta(p, a, \epsilon)$. This element will be popped by the very last step, and will correspond to a transition like $(q, \epsilon) \in \delta(s, b, t)$. These are exactly the conditions under which we have a rule in the CFG of the form $A_{p,q} \to aA_{r,s}b$. Also note that our string could be written as $x = ayb$, where y was consumed on the way from r to s (i.e. excluding the first and last steps). Now, as the automaton moved from r to s, it did so with a stack that started by containing t. But it was not allowed to touch it, so whenever it had to decide on its next move, and it was all the way down to the t on the stack, it would have chosen a push move, which is also a move that it could have chosen if the stack was empty. So in other words, the transition from r to s that the automaton took is also a transition it could have taken if it had an empty stack at this point. Since that transition has length no more than k, the inductive hypothesis tells us that there is a corresponding derivation from $A_{r,s}$ to y (y is the input consumed on going from r to s). But then, combining this with the earlier production rule, we get:

$$A_{p,q} \Rightarrow aA_{r,s}b \Rightarrow^* ayb = x$$

This completes our proof of the correspondence between the two approaches to context-free languages.